CS 610 Semester 2022–2023-I: Assignment 3

7th September 2022

Due Your assignment is due by Sep 19, 2022, 11:59 PM IST.

General Policies

- You should do this assignment ALONE.
- Do not plagiarize or turn in solutions from other sources. You will be PENALIZED if caught.

Submission

- Submission will be through mooKIT.
- Name each source file as "<roll-no-probno>.cpp" (e.g., "22111090-prob2.cpp").
- Submit a PDF file with name "<roll-no>.pdf". Describe your solutions to the problems in the PDF, and explain your implementations. Describe any challenges (if any) that you may have faced.
- Provide exact compilation instructions.
- We will provide template code for a few problems. Modify the template code as instructed and perform evaluations. Compare the performance of the different versions and report speedup results with the GNU GCC compiler.
- Include the GCC compiler version and the system description (e.g., levels of private caches, L1/L2/LLC cache size, and processor frequency) in your report.
- We encourage you to use the LaTeX typesetting system for generating the PDF file.
- You will get up to TWO LATE days to submit your assignment, with a 25% penalty for each day.

Evaluation

- Your solutions should only use concepts that have been discussed in class.
- Show your computations where feasible and justify briefly in the PDF.
- Write your code such that the EXACT output format is respected.
- We will evaluate the implementations with our OWN inputs and test cases (where applicable), so remember to test thoroughly.
- We may deduct marks if you disregard the listed rules.

Problem 1

[30 marks]

Consider the following code:

```
int i, j, k, l;
for (i = 0; i < 1024; i++) {
 for (j = 1; j < 1024; j++) {
  for (k = j; k < i; k++) {
   for (l = 1; l < k; l++) {
    S(i, j, k, l);
   }
  }
 }
}
```

The data dependences for the loop are given to be (1,0,-1,1), (1,-1,0,1), (0,0,1,-1), and (0,1,0,-1).

(a) Which loops, if any, are parallel? Why?

(b) What are the valid permutations of the loop? Why?

(c) Which loops, if any, are valid to unroll and jam? Why?

(d) What tiling is valid, if any? Why?

(e) Show code for the *jilk* form of the code. For this part, ignore the above dependences and assume *jilk* permutation is allowed.

Problem 2

[20+40 marks]

Consider the following code snippet.

```
#define N (1 << 13)

double x[N], y[N], z[N], A[N][N];

for (int i = 0; i < N; i++) {
 for (int j = 0; j < N; j++) {
  y[j] = y[j] + A[i][j]*x[i];
  z[j] = z[j] + A[j][i]*x[i];
 }
}
```

Perform loop transformations to improve the performance of the following code snippet for sequential execution on one core (i.e., no multithreading). We will provide a template code to help with the computation.

1. Modify the `optimized()` function in the template code and perform evaluations.
2. Implement a version of the best-performing variant from the previous step using AVX/AVX2 intrinsics. Implement it as a separate function.

You may use any transformation we have discussed in class. You can try to align statically/dynamically allocated memory, and are free to use constructs like `__restrict__` and `#pragma GCC ivdep`. NO ARRAY PADDING OR LAYOUT TRANSFORMATION OF ARRAYS (e.g., 2D->1D) are allowed.

Some transformations will lead to speedup, some will not. Include all the transformations that you tried, even if they did not work. Finally, summarize the set of transformations (e.g., 32 times unrolling + `xyz` permutation) that gave you the best performance.

Time the different versions and report the speedup results with the GNU GCC compiler. Note that GCC does not auto-vectorize at optimization level `-O2`, you need to pass `-O3`. Evaluate part (i) with both optimization levels `-O2` and `-O3`.

Briefly explain the reason for the poor performance of the provided reference version, your proposed optimizations (with justifications) to improve performance, and the improvements achieved.

Problem 3

[30 marks]

Consider the following code snippet.

```
#define N (1 << 12)
#define ITER 100

double A[N][N];

for (int k=0; k<ITER; k++) {
 for (int i=1; i<N; i++) {
  for (int j=0; j<N-1; j++) {
   A[i][j+1] = A[i-1][j+1] + A[i][j+1];
  }
 }
}
```

Create one or more parallel version(s) using OpenMP. Feel free to apply *valid* loop transformations to get the best performance. Report the speedup you get with your OpenMP parallel version(s).

Problem 4

[30 marks]

Assume an array of type `int`. The size of the array is 2^{24}. Implement computing the sum of the elements of an array using the following strategies.

(a) OpenMP `parallel for` version *without* reductions. You are allowed to use synchronization constructs like `critical` or `atomic`.

(b) an OpenMP `parallel for` version *using* reductions,

(c) OpenMP tasks where each task will work on a sub-array of size 1024.

You are welcome to implement additional versions for better performance.

Problem 5

[20+40+40 marks]

Implement versions of an *inclusive* prefix sum function using OpenMP, SSE4 intrinsics, and AVX/AVX2 instrinsics. Do not use AVX-512. Compare the performance with the sequential version, and report performance speedups.

Include the compiler version and the exact compilation command. It is okay to tweak the provided compilation command to suit your compiler version and the target architecture. You should refer to the GCC manual for details.

Use the template code provided with the description. There are multiple ways to compute prefix sum with OpenMP, SSE4, and AVX/AVX2. Try to come up with implementations that perform the best for you.

Refer to `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#` for documentation on Intel Intrinsics.